Why Simulation-Based Approachs with Combined Fitness are a Good Approach for Mining Spaces of Turing-equivalent Functions

Olivier Teytaud

Why Simulation-Based Approachs with Combined Fitness are a Good Approach for Mining Spaces of Turing-equivalent Functions

O. Teytaud

***Abstract*— We show negative results about the automatic generation of programs within bounded-time. Combining recursion theory and statistics, we contrast these negative results with positive computability results for iterative approachs like genetic programming, provided that the fitness combines e.g. fastness and size. We then show that simulation-based approachs (approachs evaluating only by simulation the quality of programs) like GP are not too far from the minimal time required for evaluating these combined fitnesses.**

I. Introduction

```
@inProceeding{teytrice,
author={O. Teytaud},
title={Why Simulation-Based Approachs
with Combined Fitness are a Good Approach
for Mining Spaces of Turing-equivalent
functions},
booktitle={12 pages, proceedings of CEC},
year=2006 }
```

Inspired by the genetic programming (GP) paradigm [9], [13], [15], we investigate conditions under which the automatic generation of programs is possible. Precisely, we study programs aimed at generating programs for a given target-task, where the target-task might be provided by the user to the automatic generator as a black-box or as a Turing-machine number. In this spirit, we compare results derived from recursion theory applied to finite-time computations, and results on iterative algorithms derived from statistics and optimization in a spirit close to GP:

- we show universal lower bounds on the possible efficiency of (possibly randomized) programs aimed at optimizing in finite time i) the size, ii) the time complexity or iii) the space complexity. These results hold even within arbitrary large tolerance functions allowing strong sub-optimality. The uncomputability results are in particular stronger for the size of programs (i.e. they remain in the iterative case also, what does not happen for speed), what is related to the phenomenon of bloat (see below), which is an important issue in GP.
- we then turn our attention to ”blind” algorithms, that use only the target-task as a black box, and converge iteratively as in GP. Whenever previous negative results hold for bounded-time algorithms, we show positive results in an iterative convergence sense classical in optimization. Moreover, the positive results are proved thanks to population-based methods (keeping in memory a population of programs with their associated fitnesses) very related to GP methods. We then show lower bounds on the time complexity of iterative automatic program building, that are close to the simulation cost, thanks to a modified version of Kolmogorov’s complexity ; this shows that the computational cost of simulation cannot be avoided ;
- we then show drawbacks that hold for fitnesses taking into account size alone or speed alone, and that do not hold for combined fitnesses using both ([29], [33], [17], [19]).

Équipe TAO (INRIA Futurs), LRI, UMR 8623 (CNRS - Université Paris-Sud), Bat. 490, Université Paris Sud, 91405 Orsay CEDEX, France. Contact author: olivier.teytaud@lri.fr. O. Teytaud is partially supported by the Pascal Network of Excellence.

We will need the following concepts in the paper:

- **programming-programs** are programs that output programs.
- **finite-time algorithms** take something as input, and after a finite-time (depending upon the entry), give an output. This is usually what we call an ”algorithm”. The opposite concept is **iterative algorithms**, which take something as input, and during an infinite time provide outputs, that are e.g. converging to the solution of an equation. Of course, the set of functions that are computable in finite time is included in (and different from) the set of functions that are the limit of iterative algorithms (see also [24]). The (time or space) complexity of iterative algorithms is the (time or space) complexity of *one computation* of the infinite loop with one entry and one output. Therefore, there are two questions quantifying the overall complexity: the convergence rate of the outputs to a nice solution, and the computation time for each run through the loop.
- **generalization** is the process by which a function, calibrated in order to work on some entries, work also on other entries. The study of generalization is the main topic of statistical learning. A survey can be found in [10], [32], [18].
- **genetic-programming** ([9]) is the research of a program realizing a given target-task roughly as follows:
  1) generate (at random) an initial population of algorithms ;
  2) select the ones that, after simulation, look the most

relevant for the target-task (this is dependent of a distance between the results of the simulation and the expected results, that is called the *fitness*) ;
3) create new programs by randomly combining and randomly mutating the ones that remain in the population ;
4) go back to step 2.

- **bloat** is the unexpected increase of the size of automatically generated programs. Bloat is an important issue in GP [11], [1], [14], [25], [22], [30], [2], [12].
- the **absolute** Kolmogorov complexity ([27], [28], [8]) of x is the size of the shortest program outputting x on the entry 0. We study in this paper a modified version, inspired by [3], [6], [26], which is the size of the shortest program that outputs x on entry 0 and that works in time $\leq T$. This modified version is computable, and we show lower bounds on its complexity.

II. Definitions & notations

For the sake of clarity and without loss of generality, we consider Turing-machines [31], [23] (TM) with one (read-only) input tape, where the head moves right if and only if the bit under the reading head has been read, one internal tape (read and write, without any restriction on the allowed moves), one (write-only) output tape, which moves of one and only one step to the right at each written bit. The restrictions on the moves of the heads on the input and on the output tapes do not modify the expressive power of the TMs as they can simply copy the input tape on the internal tape, work on the internal tape and copy the result on the output tape. The space complexity is with respect to the internal tape (number of visited elements of the tape) *plus* the size of the program. All tapes' alphabets are binary. These Turing-machines can work on rational numbers, encoded as 2-uples of integers. Thanks to the existence of Universal Turing Machine, we identify TM and natural numbers in a computable way (one can simulate the behavior of the TM of a given number on a given entry in a computable manner). We use capital letters for programming-programs, i.e. programs that are aimed at working on programs. If x is a program and e an entry, then $x(e)$ is the output of the application of x to the entry e. $x(e) = \perp$ is the notation for the fact that x does not halt on entry e. We also note $\perp$ a program such that $\forall e; \perp (e) = \perp$. A program p is a total computable function if $\forall e \in \mathbb{N}; p(e) \neq \perp$. A decider is a total computable function with values in $\{0, 1\}$. We note D the set of all deciders. We say that a function f recognizes a set F among deciders if and only if $\forall e; (e \in F \cap D \rightarrow f(e) = 1$ and $e \in D \setminus F \rightarrow f(e) = 0)$ (whatever may be the behavior, possibly $f(e) = \perp$, for $e \notin D$). We say that two programs x and y are equivalent if and only if $\forall e \in \mathbb{N}; x(e) = y(e)$. We note this $x \equiv y$. We note $\equiv_y = \{x; x \equiv y\}$. We note $\mathbf{1} = \{p; \forall e, p(e) = 1\}$. The definition of the size $|x|$ of a program x is any usual definition such that the number of states is upper-bounded by an increasing computable function of the size. We note (with a small abuse of notation as it depends on f and x and not only on $f(x)$) $time(f(x))$ (resp. $space(f(x))$) the computation time (resp. the space complexity) of program f on entry x. We note $< x_1, \ldots, x_n >$ a n-uple encoded as a unique number thanks to a given recursive encoding.

$\mathbb{E}$ is the expectation operator. $Proba(.)$ is the probability operator ; by abuse, depending on the context, it is sometimes with respect to (x, y) and sometimes with respect to a sample $(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_m)$. Iid is a short notation for "independent identically distributed".

Overview of the paper

Section III presents uncomputability results for finite-time algorithms. Section IV presents mainly computability results for the specialization on finite samples ; this section provides building blocks for section V which shows positive results for GP-like iterative algorithms. Section VI concludes.

III. Finite time algorithms

We consider the existence of programs $P(.)$ such that:

- the user provides x, which is a Turing-computable function ;
- $P(x) = y$, where $y \equiv x$ and y is not too far from being optimal (for size, space or time).

Theorem 1 shows that for reasonable formalizations of this problem, such programs do not exist. This result is an extension of classical uncomputability examples (the classical case is $C(a) = a$).

Theorem 1 (Undecidability): Whatever may be the function $C(.)$ in $\mathbb{N}^{\mathbb{N}}$, there does not exist P such that for any total function x, $P(x)$ is equivalent to x and $P(x)$ has size $|P(x)| \leq C(\inf_{y \equiv x} |y|)$.

Moreover, for any $C(.)$, for any such non-computable $P(.)$, there exists a Turing-machine using $P(.)$ as oracle, that solves a problem in $0'$, the jump of the set of computable functions.

Proof: Assume, in order to get a contradiction, that such a $P(.)$ exists.

Step 1: we study the behavior of $P(.)$ on $\mathbf{1}$.

Then, define y as the shortest program such that $y(e) = 1$ for any entry e, and $Y = \{z; z \equiv y$ and $|z| \leq C(|y|)\}$. Y is finite.

Then, consider a program x that always halts. Necessarily, $P(x) \in Y$ if and only if $x \equiv y$.

Step 2: show that thanks to P (if it exists), we can decide $\mathbf{1}$ (the class of programs that always reply 1) among deciders (indeed, more generally among computable total functions).

As Y is recursive (as it is finite), there exists a program Q such that $Q(e) = 1$ if $e \in Y$ and $Q(e) = 0$ otherwise.

Therefore, thanks to step 1, $R = Q \circ P$ has the following property for any x that always halts: $R(x) = 1$ whenever $x \equiv y$ and $R(x) = 0$ in other cases

Step 3: we now show that recognizing $\mathbf{1}$ among programs that always halt is as difficult as the domain-emptyness problem (that is known undecidable since Turing's fundamental paper [31]). Formally, we show that with an oracle recognizing $\mathbf{1}$ among deciders, there exists a Turing-machine only feeding the oracle with deciders that recognizes $\equiv_\perp$.

Consider the following program S working on entry $< x, < a, b >>$:

- simulate a steps of x on entry b ;
- if it halts during this simulation then reply 0.
- if it does not halt, reply 1.

This program S always halts.

Then consider the following program working on entry x, using an oracle $R(.)$. It recognizes $\mathbf{1}$ among deciders:

- if $R(k \mapsto S(x,k)) = 1$, then reply 1.
- otherwise, reply 0.

This program replies 1 if and only if x never halts on any entry. Therefore, this program solves the emptyness of the domain of a Turing-machine (it recognizes $\equiv_\perp$). This is known as an uncomputable task, and more precisely it is in $0'$ (the jump of the set of Turing-computable problems). Therefore, we have shown that no computable $R(.)$ recognizing $\mathbf{1}$ among deciders can exist. As step 2 shows that the existence of a suitable computable $P(.)$ implies the existence of such a computable $R(.)$, such a computable $P(.)$ does not exist. □

We now show that using a random generator does not change the result.

Corollary 2 (No size optimization): Whatever may be the function $C(.)$, there does not exist any program P, even possibly using a random oracle providing independent random values uniformly distributed in $\{0,1\}$ such that for any total function x, with probability at least $2/3$, $P(x)$ is equivalent to x and $P(x)$ has size $|P(x)| \leq C(\inf_{y \equiv x} |y|)$.

Proof: We only simulate all the possible runs and modify the decision method in step 2 of the previous proof.

In the new second step, we simulate on a Turing machine, simultaneously[1], all the possible behaviors of P until we reach a total probability $> \frac{1}{2}$ of halting with $P(x) \in Y$ or a probability $> \frac{1}{2}$ of halting with $P(x) \notin Y$. One of these two cases must necessarily occur by definition of P. □

The extension from size of programs to time complexity of programs requires a more tricky formulation than a simple total order relation "is faster than" ; a program can be faster than another for some entries and slower for some others. A natural requirement is that a program that suitably works provides a (at least nearly) Pareto-optimal program [20], i.e. a program f such that there's no program that is as fast as f for all entries, and better than f for some specific entry, at least within a tolerance function $C(.)$. The precise formulation that we propose is somewhat tricky but indeed very general:

Corollary 3 (Time complexity): Whatever may be the function $C(.)$, there does not exist any program P, even possibly using a random oracle providing independent random values uniformly distributed in $\{0,1\}$, such that for any total function x, with probability at least $2/3$,

$P(x) \equiv x$ and there's no $y \equiv x$ such that y Pareto-dominates $P(x)$ (in time complexity) within $C(.)$, i.e. $\not\exists y \in \equiv_x$ and

$$\forall z; time(P(x)(z)) \geq C(time(y(z)))$$

$$\text{and } \exists z; time(P(x)(z)) > C(time(y(z)))$$

The result is also true when restricted to x such that a Pareto-optimal function exist.

Proof: The proof is very similar to the previous proof. The only Pareto-optimal time complexity for $\mathbf{1}$ is a constant K (the time required to output 1 in the chosen encoding).

Therefore, for any entry $x \in \mathbf{1}$, P must generate a program in Y, where Y is the class of programs always outputing 1 and halting within time complexity $\leq C(K)$.

Y is not finite, but is recursive (lemma below). Within this modification, steps 2 and 3 of the proof of theorem 1 still hold. □

We now prove the following lemma, useful in the proof above.

Lemma 4 (Computability for bounded-time): For any $(k, C) \in \mathbb{N}$, the set of computable functions f such that $\forall x; time(f(x)) \leq k$ and $\forall x; f(x) = C$ is computable.

Proof: Consider the program that works as follows on a program p:

- write the tree of all the possible runs within the k first steps.
- if at least one of these runs does not halt withint the k steps, then reply "no".
- if at least one of these runs replies something else than C, then reply "no".
- otherwise else, reply "yes". □

After size (corollary 2) and time (corollary 3), we now consider space complexity (5):

Corollary 5 (Space complexity): Whatever may be the function $C(.)$, there does not exist any program P, even possibly using a random oracle providing independent random values uniformly distributed in $\{0,1\}$, such that for any total function x, with probability at least $2/3$,

$P(x) \equiv x$ and there's no $y \equiv x$ such that y dominates $P(x)$ (in space complexity) within $C(.)$, i.e., $\not\exists y, y \equiv x$ and

$$\forall z; space(P(x)(z)) \geq C(space(y(z)))$$

$$\text{and } \exists z; space(P(x)(z)) > C(space(y(z)))$$

Proof: The proof is very similar to the two previous ones. We consider the same target-task (i.e. always writing 1 on the output tape). This can be performed within constant space complexity S. If such a P exists, then it must write, with probability at least $2/3$, a program in Y, where Y is the class of programs writing 1 within space complexity $C(S)$. This class is computable (lemma below), so steps 2 and 3 of the proof of theorem 1 hold within this modification. □

Remark 6 (Other fitnesses): We have proved the non-computability result for speed, size and space. Other fitnesses (in particular, mixing these three fitnesses) lead to the same

[1] By "simultaneously" we mean that we simulate all the possible runs simultaneously, in a breadth-first manner.

result. The key of the proofs above (th. 1, corollaries 2, 3, 5) is the recursive nature of sets of functions optimal for the given fitness, which is a very stable feature.

We now prove the following lemma, useful in the proof of the previous corollary 5.

Lemma 7 (Computability for bounded space): For any $(k, C) \in \mathbb{N}$, the set $\mathcal{S}$ of computable functions f such that $\forall x; space(f(x)) \leq k$ and $\forall x; f(x) = C$ is computable.

Proof:

- we recall that our definition of space complexity includes the size of the program. Therefore, TMs with a bounded space complexity have a bounded number of configurations[2] ; they are finite automata in which some nodes have an output (recall that our TMs have restrictions on the possibility of moves of heads on the input and output tapes). Note A_f such a finite automaton, associated to a TM f.
- note $Q = \{q_1, q_2, \ldots, q_N\}$ the finite set of states of A_f, some of them being halting states and some of them outputing 1, some of them outputing 0, some of them not outputing anything, some of them reading the bit under the input head and some others not. Assume without loss of generality that q_1 is the initial state.
- define a new automaton A'_f on the set of states $Q \times \cup_{l=0}^{k}(\{0,1\}^l)$ where k is the length of C.
- set the initial state of A'_f at $(q_1, \#)$ where $\#$ is the empty string.
- define the transitions of A'_f as follows: there is a transition from state $(q_i, S_{i'})$ (where $S_{i'}$ is a binary string) to $(q_j, S_{j'})$, with $S_{j'} = S_{i'}.b$ (where . denotes the concatenation operator) when reading entry e on the input tape (possibly e is the empty string if A_f does not read the input tape at state q_j), if and only if A_f has a transition from q_i to q_j when reading e and A_f outputs b (possibly the empty character if there's no output) in this case. note that A'_f is a finite automaton without any writing hability.
- set the initial state at (q_1, S_0) where S_0 is the empty string.
- then, f is in $\mathcal{S}$ if and only:
  - f can be consistently translated to A'_f as explained above, and
  - A'_f halts in $Q \times \{C\}$ on any entry.

  both these statements are decidable, therefore $\mathcal{S}$ is decidable. □

IV. Specialization on a finite sample

We now turn our attention to the specialization on a finite sample. Results below will be used as building blocks for theorems of section V about iterative algorithms.

Theorem 8 (Specialization on a finite sample): a) There exists a program P such that $\forall m \in \mathbb{N}, \forall i \in [[1, m]], P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)(x_i) = y_i$ and $P(< x_1, \ldots, y_m >)(x_i) = y_i$ has optimal average (on the x_i) time complexity (resp. space complexity) , i.e. $\frac{1}{n}\sum_{i=1}^{n} time(P(< x_1, \ldots, y_m >)(x_i))$ (resp. $\frac{1}{n}\sum_{i=1}^{n} space(P(< x_1, \ldots, y_m >)(x_i))$) minimal.

b) There does not exist a program P such that $\forall m \in \mathbb{N}, \forall i \in [[1, m]], P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)(x_i) = y_i$ and $P(< x_1, \ldots, y_m >)$ has optimal size (i.e. $|P(< x_1, \ldots, y_m >)|$ minimal).

(we assume for consistency that $x_i = x_j$ implies $y_i = y_j$; we consider that the program P is right provided that it works in this safe case, whatever may be its behavior in other cases)

c) For any $(c_1, c_2, c_3) \mapsto c(c_1, c_2, c_3)$, non-decreasing computable function with limit $+\infty$ as a function of c_1 or as a function of c_2, there exists a program P such that $\forall m \in \mathbb{N}, \forall i \in [[1, m]], P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)(x_i) = y_i$ and P has optimal average (on i) cost, where the cost of program p on entry e is $c(time(p(e)), space(p(e)), |p|)$.

Remark 9 (Fast programs do not generalize well): Note that $a)$ in the case of time complexity is only of theoretical importance as Turing-machines optimal for time complexity on a finite set of cases are essentially very big Turing machines outputting the y_i as soon as x_i is recognized through a full-branching reading process. These machines work on the $(x_i, y_i)_{i \leq m}$ but not necessarily on unseen (x, y) (no *generalization* ability). The adaptation to c) is more concrete, as shown by theorems 10 and 11 below.

Proof of the theorem:

a) is realized by the following program in the case of time complexity:

- compute the maximal time complexity T of the naive program comparing an input e to each of the x_i, and replying y_i if $e = x_i$ and replying 0 in other cases.
- consider the programs of time complexity bounded by this time complexity. Simulate all of them within T steps ; there are infinitely many such functions, but we only have to take into accound the K states that can be reached within T steps, where K is the maximal number of states that can be visited by a program of time complexity $\leq T$ (for our Turing-machine formalism, $K \leq 4^T$ with binary tapes, as 2 binary values are read (one on the input tape and one on the internal tape).
- simulate all of them until step T on all entries.
- select one of them which is optimal from the point of view of the average time complexity.

The case of space complexity is similar.

b) can be proved by the following reduction:

- assume that such a P exists ;
- consider the program $e \mapsto |P(\text{empty string}, e)|$;
- this programs computes the absolute Kolmogorov complexity[27], [28], [8], [4], [5], what is not possible.

c) is derived as a) ; the properties of $c(.,.,.)$ ensures that a finite set of functions can be considered (either the set of functions with bounded time complexity if $c(., 0, 0) \to \infty$, which is finite is we restrict our attention to the finitely many possible time steps of simulation as in part a) of this proof, or

[2]Here, a configuration is the current state plus the state of the internal tapes.

functions with bounded space complexity if $c(0,.,0) \to \infty$). □

Note that we have not ensured that the resulting program halts within the same time (resp. space) complexity on other entries than the x_i, y_i for $i \in [[1,m]]$. We now have to prove that working on a sample might be efficient in generalization (ensuring that $P(< x_1, \ldots, y_m >)$ halts on any x and gives the right answer with probability 1, at least if m is sufficiently large). This is a problem of *statistical learning* (see e.g. [32], [10]). The usual general framework of statistical learning is as follows:

- Consider $(x_1, y_1), (x_2, y_2), \ldots$, a sequence of iid (independent identically distributed) elements of $\mathbb{N}^2$, with common law P. (x_i, y_i) is called an example. Restricting our attention to the case in which we work on consistent examples of a deterministic relation, we here consider that $P(.)$ is such that for some total computable function f, $P(f(x) = y) = 1$.
- Consider g, a function taking as input $< x_1, y_1, \ldots, x_m, y_m >$ and where $g(< x_1, y_1, \ldots, x_m, y_m >)$ is itself a Turing machine.
- Then, the so-called error rate of g after m examples $(x_1, y_1), \ldots, (x_m, y_m)$ is the probability for $P(x, y)$ of $g(< x_1, y_1, x_2, y_2, \ldots, x_m, y_m >)(x) \neq y$. It is a random variable, as it depends on the m first examples. Statistical learning theory is the study of properties of various functions g, depending (or not) upon properties of P.

Many tools exist for studying such problems. The main question is a problem of generalization: finding a function that works on $(x_i, y_i)_{i \leq m}$ is easy (i.e. it is easy to design g such that $\forall i < m, g(< x_1, y_1 >, \ldots, < x_m, y_m >)(x_i) = y_i$), but does this function generalizes well to P ? A direct proof is possible in the current framework:

Theorem 10 (Learning from deterministic examples): Assume that $y = f(x)$ with probability 1, where f is a computable function that always halts. Then, if $(x_1, y_1), \ldots, (x_m, y_m)$ is an iid sample with the same law as (x, y), then

$$Proba(P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)(x) \neq y) = 0$$

for m sufficiently large, almost surely in the sequence $(x_1, y_1), \ldots, (x_m, y_m), \ldots$, whenever $f = P(< x_1, \ldots, y_m >)$ is the first (for any enumeration of functions) computable function such that $\forall i \in [[1,m]]; f(x_i) = y_i$.

In this theorem we do not here assume (and do not conclude) that $P(.)$ is computable. This will be done in the next section after a slight modification of the paradigm. Note that the use of an order independent of the target-task has been investigated in GP (lexicographic order, see [17]).

Proof: Consider f the first function such that $Proba(f(x) \neq y) = 0$.

For any $g < f$, $Proba(g(x) \neq y) > 0$. Therefore, for any $g < f$, almost surely, there exists $i_g < \infty$ such that $g(x_{i_g}) \neq y_{i_g}$.

As the set $\{g; g < f\}$ is finite, the previous sentence can be rewritten: almost surely, for any $g < f$, there exists $i_g < \infty$ such that $g(x_{i_g}) \neq y_{i_g}$.

For $m > \sup_{g<f} i_g$, the property $Proba(P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)(x) \neq y) = 0$ holds. □

The previous theorem holds for any $P(.)$ verifying the required properties. Indeed, even if the order is computable, $P(.)$ is not necessarily computable (e.g. it is not for an ordering by size). On the other hand, as stated in theorem 8, $P(.)$ is computable if the order is the average time. Unfortunately, this is not in the scope of theorem 10, as the order depends on the examples whereas we need in theorem 10 an order that is not dependent on the data. Fortunately, the following theorem combines the advantages of both theorems 8 and 10: it provides a fitness such that $P(.)$ is computable and generalization holds.

V. Iterative algorithms

We have shown in section III that finite-time algorithms have deep limits. We have shown in section IV that iterative paradigms could converge to nice solutions (i.e. solutions that generalize well). We now have to proof that that iterative paradigms can be implemented on a Turing machine.

So, we now show in theorem 11 that the limit behavior of iterative paradigms can be reached by Turing-computable iterative algorithms. Theorem 12 is a refinement from the point of view of complexity.

The following theorem deals with learning deterministic computable relations from examples.

Theorem 11: Assume that $y = f(x)$ where f is computable and $Proba(f(x) = \perp) = 0$ and $\mathbb{E}time(f(x)) < \infty$. Then, if $(x_1, y_1), \ldots, (x_m, y_m)$ is an iid sample with the same law as (x, y), then

$$Proba(P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)(x) \neq y) = 0$$

for m sufficiently large, almost surely. whenever $f_m = P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)$ is a computable function f_m such that $\forall i; f_m(x_i) = y_i$, minimal for the criterion $c(ACT_m(f_m), |f_m|)$, where $ACT_m(g)$ is the average computation time of g on the x_i and $c(a,b)$ is any computable function, continuous and increasing as a function of a (which is rational) and increasing as a function of b, such that $\lim_{a\to\infty} c(a,0) = \lim_{b\to\infty} c(0,b) = \infty$.

Moreover, $c(\mathbb{E}time(f_m(x)), |f_m|)$ converges to the optimal limit:

$$c(\mathbb{E}time(f_m(x)), |f_m|)$$
$$\to \inf_{f; Proba(f(x)\neq y)=0} c(\mathbb{E}time(f_m(x)), |f|)$$

and f_m is computable from $< x_1, y_1, \ldots, x_m, y_m >$.

Proof: 1. Note f^* an unknown computable function such that $Proba(f^*(x) \neq y) = 0$, with $\mathbb{E}time f^*(x)$ minimal.

2. The average computation time of f^* on the x_i converges almost surely (by the strong law of large numbers). Its limit is dependent of the problem ; it is the expected computation time of f^* on x.

3. By definition of f_m and by step 2, $f_m = P(< x_1, \ldots, x_m, y_1, \ldots, y_m >)$ is such that $c(ACT_m(f_m), |f_m|)$ is upper bounded by $c(ACT_m(f^*), |f^*|)$, which is itself almost surely bounded above as it converges almost surely (Kolmogorov's strong law of large numbers [7]).

4. Therefore, f_m, for m sufficiently large, lives in a finite space of computable functions $\{f; c(0, |f|) \leq c(\sup_i ACT_i(f^*), |f^*|)\}$.

5. Consider g_1, ..., g_k this finite family of computable functions.

6. Almost surely, for any $i \in [[1, k]]$ such that $Proba(g_i(x) \neq y) > 0$, there exists m_i such that $g_i(x_{m_i}) \neq y_{m_i}$. These events occur simultaneously as a finite intersection of almost sure events is almost sure ; so, almost surely, these m_i all exist.

7. Thanks to step 6, almost surely, for $m > \sup_i m_i$, $Proba(f_m(x) \neq y) = 0$.

8. Combining 5 and 7, we see that $f_m \in \arg\min_G c(ACT_m(g), |g|)$ where $G = \{g_i; i \in [[1, k]] \text{ and } Proba(g_i(x) \neq y) = 0\}$.

9. $c(ACT_m(g_i), |g_i|) \rightarrow c(\mathbb{E}time(g_i(x)), |g_i|)$ almost surely for any $i \in [[1, k]] \cap \{i; \mathbb{E}time(g_i(x)) < \infty$ as $[[1, k]]$ as $c(.,.)$ is continuous with respect to the first variable (Kolmogorov's strong law of large numbers). As this set of indexes i is finite, this convergence is uniform in i.

10. $c(ACT_m(g_i), |g_i|) \rightarrow \infty$ uniformly in i such that $\mathbb{E}time(g_i(x)) = \infty$ as this set is finite.

10. Thanks to steps 8 and 9, $c(\mathbb{E}time(f_m(x)), |f_m|) \rightarrow \inf_{g; Proba(g(x) \neq y) = 0} c(\mathbb{E}time(g(x)), |g|)$. □

We now turn our attention to a slightly modified definition of f_m, which has the advantage of being more quickly computable. For the sake of clarity, the complexity below is with respect to a stronger form of machines, that can simulate n steps of machine x on entry e in time $O(n)$, and computes $\times$, $+$, $/$, $-$ in $O(1)$.

The following theorem deals with the complexity of learning Turing computable relations from examples.

Theorem 12: Assume that $Proba(g(x) \neq y) = 0$ for some computable g, and that $\mathbb{E}time(g(x))$ is finite.

Consider a Turing machine that works with an oracle tape providing a new example x_m, y_m at each request, and outputs f_m on the output tape. The Turing machine works in-line, i.e. provides a new f_m at each request on the oracle.

Then, within logarithmic factors or computational costs associated to the computation of a finite number of calls to $c(.,.)$, the following algorithm works with asymptotic time complexity $O(L(m))$ where $L(.)$ is a non-decreasing computable function such that $time(L(m)) = O(L(m))$ (e.g. $\log_2(.)$.

Define $\Delta_0 = 1$. Define $t(f, 0) = 0$ and for $m \geq 1$ $t(f, m) = \frac{1}{m}((m-1) \times t(f, m-1) + min(time(f(x_m)), L(m)))$. Define Pop_0 the empty population. Define f_0 a function that just outputs 1 and halts. Define $\forall x; fit_0(x) = 0$. At each new example x_m, y_m ($m \geq 1$):

- set $Pop_m = \{f; c(0, |f|) \leq \min(\Delta_m, fit_{m-1}(f_{m-1}))\} \cup Pop_{m-1}$.
- for $f \in Pop_m$, define $fit_m(f) = c(t(f, m), |f|)$.
- for functions $f \in Pop_m$ which finish in time $\leq L(m)$ and reply some $f(x_m) \neq y_m$, then set $fit_m(f) = \infty$.
- define $f_m \in \arg\min_{Pop_m} fit_m$
- if $fit_m(f_m) > \Delta_{m-1}$, then set $\Delta_m = \Delta_{m-1} + 1$; otherwise, $\Delta_m = \Delta_{m-1}$.

Then $L(f_m) \rightarrow \inf_{f \equiv g} L(f)$ with $L(f) = c(\mathbb{E}time(f(x)), |f|)$, and almost surely the time complexity per value of m is $O(L(m))$.

An interesting point is that the proof involves an algorithm with a population of functions with their fitnesses in memory, what is very close to genetic programming. **Proof:** The steps of the proof are as follows:

- define $f^* = \arg\min_{\{f; f \equiv g\}} L(f)$.
- define $F = \cup_{m \in \mathbb{N}} Pop_m$.
- Assume, in order to get a contradiction, H1: F does not contain any f such that i) $Proba(f(x) = y) = 1$ and ii) f has finite expectation time.
- Then F is finite, otherwise else $\Delta_m \rightarrow \infty$ and $\inf_F fit_m \rightarrow \infty$ and therefore f^* is in F.
- Then, all $f \in F$ have $fit_m(f) \rightarrow \infty$. Proof: for each $f \in F$,
  - either there exists a, b such that $P(x = a, y = b) > 0$ and $f(a) \neq b$, and a, b will be drawn infinitely often, and in particular at some value of m for which $L(m) \geq time(f(a))$; in this case, $fit_m(f)$ reaches infinity.
  - or f does not have a finite expectation time, and $fit_m(f) \rightarrow \infty$ by the lemma below (case 2).
- therefore, all fitnesses in F run to infinity. This leads to $\Delta_m \rightarrow \infty$. This implies that $f^* \in F$; this is a contradiction with H1. Therefore, H1 does not hold ; for m sufficiently large (say $m \geq m_0$), some $f \in Pop_m$ verifies $Proba(f(x) = y) = 1$ and f has finite expectation time.
- then, for $m \geq m_0$, $fit_m(f) \leq fit_m(f^*) \leq K$ for some $K > 0$ as $fit_m(f^*)$ converges (cf lemma below, case 1).
- therefore, Δ_m is also bounded above. Hence, F is finite.
- applying again the lemma below, we see that $fit_m(f)$ converges to $L(f)$ uniformly in $f \in F$ with finite expectation time and $Proba(f(x) = y) = 1$, and converges to infinity uniformly for other $f \in F$. This uniform convergence implies that $L(f_m) \rightarrow L(f^*)$. □

Lemma 13 (Adapted strong law of large numbers): 1. Define $L(m) \rightarrow \infty$ as $m \rightarrow \infty$. Assume that x is a non-negative random variable with finite expectation. Then $e_m = \frac{1}{m} \sum_{i=1}^{m} \min(x_i, L(i)) \rightarrow \mathbb{E}x$, if the x_i are an iid sample with the same law as x.

2. Assume that x has infinite expectation. Then $e_m = \frac{1}{m} \sum_{i=1}^{m} \min(x_i, L(i)) \rightarrow \infty$.

Proof of 1 (2 is similar):

Step 1: upper bound. $e_m \leq m_m = \frac{1}{m}\sum_{i=1}^{m} x_i$ and by the strong law of large numbers from Kolmogorov the right term goes to $\mathbb{E}x$.

Step 2: lower bound. $m_m - e_m = \frac{1}{m}\sum_{i=1}^{m} \max(0, x_i - L(i))$. For any $m > m_0 \geq 1$, the right-hand-term is upper bounded by

$$\underbrace{\frac{1}{m}\sum_{i=1}^{m_0} \max(0, x_i - L(i))}_{\to 0}$$

$$+ \underbrace{\frac{m-m_0}{m}}_{\to 1} \underbrace{\frac{1}{m}\sum_{i=m_0+1}^{m} \max(0, x_i - L(m_0))}_{\to K(m_0)}$$

where the right-hand-term convergence is Kolmogorov's strong law of large numbers [21], [7], with $K(m_0)$ is the expectation $\mathbb{E}\max(0, x - L(m_0))$. We now show that $K(m_0)$ goes to 0 as $m \to \infty$:

$$K(m_0) = \mathbb{E}\max(0, x - L(m_0))$$
$$= \sum_{n \in \mathbb{N}} \max(0, n - L(m_0)) Proba(x = n)$$

which goes to 0 by the monotone convergence theorem of Lebesgue [16].

Step 3: summary. So, for any m_0, we can sum up previous steps by the fact that almost surely

$$e_m \leq \underbrace{m_m}_{\to \mathbb{E}x} \text{ and}$$

$$e_m \geq \underbrace{m_m}_{\to \mathbb{E}x} - (1 + o(1)) \times \underbrace{\frac{1}{m}\sum_{i=1}^{m} \max(0, x_i - L(i))}_{\to_{m\to\infty} K(m_0) \to_{m_0 \to \infty} 0}$$

the second inequality holding for m sufficiently large.

Step 4: concluding. Therefore, for any ϵ,

- upper bound: for m sufficiently large, $e_m \leq \mathbb{E}x + \epsilon$ (first inequality in step 3),
- lower bound:
  - for m sufficiently large, $m_m \geq \mathbb{E}x - \epsilon/3$,
  - with m_0 such that $K(m_0) < \epsilon/3$ and for $m \geq m_0$ sufficiently large, $(1+o(1)) \times \frac{1}{m}\sum_{i=1}^{m} \max(0, x_i - L(i)) \geq K(m_0) + \epsilon/3$

  and therefore for m sufficiently large $e_m \geq \mathbb{E}x - \epsilon/3$. □

We have shown that finite time algorithms could not work properly (theorem 1, corollaries 2,3,5, remark 6). We have shown that iterative methods designing programs that are as fast as possible do not generalize well (remark 9), and that iterative methods designing programs that are as small as possible are not Turing-computable (theorem 8). We have also shown that iterative methods combining size and speed are Turing-computable and generalize well (theorem 10, 11). The complexity of Turing-computable programs defined therein is mainly the cost of simulation. We now show that it is not possible to avoid the complexity of simulation.

We first define a modified version of the complexity of Kolmogorov. We recall that Kolmogorov's complexity was first defined by Solomonov [27] in the field of artificial intelligence also. Many other works about Kolmogorov's complexity exist, in particular adding constraints on resources ([3], [6], [26]); as far as we know, the following result is different and new (more closely related to the subject of this paper).

Definition 14 (Kolmogorov's complexity in bounded time): An integer x is T,S-complex if there is no Turing machine M such that $M(0) = x \wedge |M| \leq S \wedge time(M(0)) \leq T$. M is the T-time-reduction of x if and only if $M(0) = x$, $time(M(0)) \leq T$, $|M|$ is minimal among possible functions (and, for the sake of unicity, M is the first in lexicographic ordering among M's with the same size). Consider an algorithm A deciding wether an integer x is T, S-complex or not. Define $C(T, S)$ the worst-case complexity of this algorithm ($C(T, S) = \sup_x time(A(< x, T, S >))$). It is finite for some $A(.)$, even if there's no limit on the size of x as if x is too large, it is T_n,S_n-complex whatever may be its value. We restrict our attention to such $C(.,.)$, corresponding to algorithms with computation time only depending upon T and S.

These notions are computable, but we will show that their complexity is large. The complexity of the optimization of the fitness in theorem 4 is larger than the complexity $C(.,.)$ of deciding if x is T,S-complex ; therefore, we will lower bound $C(.,.)$.

Theorem 15 (The complexity of complexness): Consider now T_n and S_n, computable increasing sequences of integers computable in time $Q(n)$ where Q is polynomial, and y_n the smallest integer that is T_n,S_n-complex. Then, for some $S_n = O(\log(n))$,

$$C(T_n, S_n) > (T_n - Q(n))/P(n).$$

where $P(n)$ is a polynomial, and in particular if T_n is $\Omega(2^n)$,

$$C(T_n, S_n) > \frac{T_n}{P'(n)}$$

where $P'(.)$ is a polynomial, i.e. essentially we can not get rid of the computation time T_n.

The proof follows the lines of the proof of the non-computability of Kolmogorov's complexity by the so-called "Berry's paradox", but with complexity arguments instead of computability arguments.

Proof:

Step 1: y_n is T_n,S_n-complex, by definition.

Step 2: But it is not $Q(n) + y_n \times C(T_n, S_n), C + D\log_2(n)$-complex, where C and D are constants, as it can be computed by

- computing T_n and S_n (in time $Q(n)$)
- iteratively testing if k is T_n, S_n-complex, where $k = 1, 2, 3, \ldots, y_n$ (in time $y_n \times C(T_n, S_n)$.

Step 3: $y_n \leq 2^{S_n}$, as:

- there are at most 2^{S_n} programs of size $\leq S_n$,

- therefore there are at most 2^{S_n} numbers that are not T_n, S_n-complex.
- therefore, at least one number in $[[0, 2^{S_n}]]$ is T_n, S_n-complex.

Step 4: if $S_n = C + D\log_2(n)$, then y_n is upper bounded by a polynomial $P(n)$ (thanks to step 3).

Step 5: combining steps 1 and 2, $y_n C(T_n, S_n) > T_n - Q(n)$.

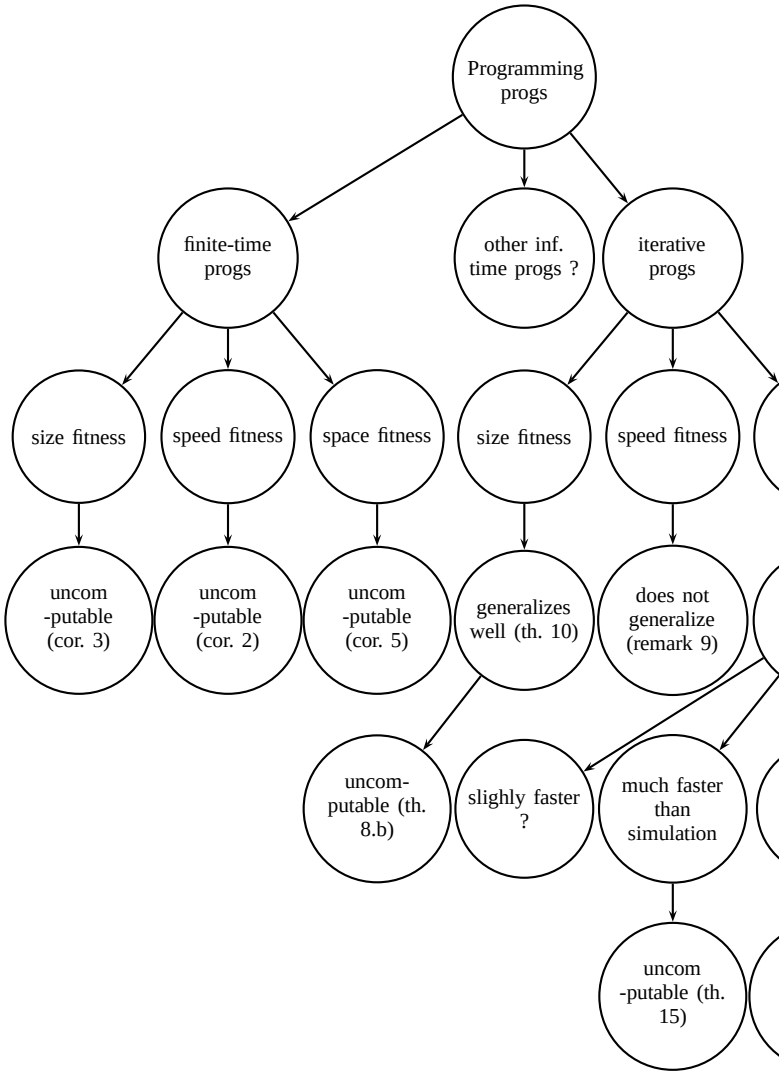
Step 6: using step 4 and 5, $C(T_n, S_n) > (T_n - Q(n))/P(n)$, hence the expected result. □

VI. Conclusion

Let's now sum up and compare our results.

- in corollaries 2, 3, 5 we have shown that finite-time programming-programs can not perform the required task, i.e. finding the most efficient function in a space of Turing-equivalent functions.
- in theorem 12, we have shown that an iterative programming-program could asymptotically perform the required target-task. E.g., GP is such an iterative method. Theorem 8 also shows that mixed fitnesses should be used ; this is very related to *bloat*, a well known problem in GP: without parsimony pressure, very long programs appear and the optimization does not work.
- GP is simulation-based and many GP-applications use fitnesses as required according to our results, i.e. mixing both size and precision.
- The main drawback of GP is that GP is slow ; one can not get rid of the computation time. In theorem 15, using a modified form of Kolmogorov's complexity, we have shown that getting rid of the simulation time is anyway not possible.

The summary of this paper, which is relevant for the research of Turing-computable functions realizing a given mapping, is as follows:



Ie, finite-time programs can not do the job (finding an optimal function in a space of Turing-equivalent functions) in the general case (corollaries 1, 2, 3). Iterative programs can do it, but only with mixed fitnesses (theorem 2b, remark 2). Then, the time-complexity of such fitness-optimization can not get rid of the simulation time (theorem 6).

In section III, we show negative results for the task consisting in writing in finite time a program realizing a given target-task in a neary optimal manner. These results are true for deterministic programming-programs, randomized programs, working on an oracle describing the target-task under the form of a program or a black-box oracle as well. These results concern time complexity, space complexity, and program size, and are true even within arbitrary tolerance functions $C(.)$.

In section V, we show positive result for the specialization on a finite sample. Learning on a finite sample with a pragmatic compromise between length and speed leads to a function which is equivalent to the real one, and which is

optimal for this compromise.

We can conclude as follows when the "target" function is a computable one:

- for various criterions (size, time complexity, space complexity), it is not possible to have a finite-time procedure that takes as input a program, and automatically generates an optimal program.
- the size is the most undecidable criterion, as even on a finite sample it remains undecidable. This is related to the so-called "bloat" phenomenon [11], [1], [14], [25], [22], [30], [2], [12].
- it is also not possible to do it from examples, and to assert after finitely many examples that the work is done and that the optimal function is found.
- on the other hand, it is possible to *converge* to a function that will match all future examples. Moreover, the resulting function will have its value, for a compromise between speed and size, optimal.

A remarkable fact is that the positive results occur for algorithms ignoring the internal structure of the program. This is a deep argument in favor of genetic programming. Note that our results also show that optimizing speed alone is not suitable on a finite sample of x_i, y_i, as very big naive programs are very fast, and that optimizing size alone is not Turing-computable ; compromises between size and time are more suitable. This is in favor of coupled fitnesses ([29], [33], [17], [19]). Another remarkable fact in favor of GP is that the simulation time is unavoidable for optimizing the relevant class of fitnesses.

This only concerns the general framework of designing Turing-computable functions. Of course, on restricted framework, automatic programing e.g. from specifications is possible.

REFERENCES

[1] Wolfgang Banzhaf and William B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, 2002.

[2] Tobias Blickle and Lothar Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms Workshop at KI-94*, pages 33–38. Max-Planck-Institut für Informatik, 1994.

[3] Harry Buhrman, Lance Fortnow, and Sophie Laplante. Resource-bounded kolmogorov complexity revisited. *SIAM Journal on Computing*, 2001.

[4] G.J. Chaitin. On the length of programs for computing finite binary sequences. *J. Assoc. Comput. Mach., 13, 547-569*, 1966.

[5] G.J. Chaitin. On the length of programs for computing finite binary sequences : statistical considerations. *J. Assoc. Comput. Mach., 16, 145-159*, 1969.

[6] L. Fortnow and M. Kummer. Resource-bounded instance complexity. *Theoretical Computer Science A*, 161:123–140, 1996.

[7] A. Y. Khintchine. Sur la loi forte des grands nombres. *Comptes Rendus de l'Academie des Sciences*, 186, 1928.

[8] A.N. Kolmogorov. Logical basis for information theory and probability theory. *IEEE trans. Inform. Theory, IT-14, 662-664*, 1968.

[9] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[10] G. Lugosi L. Devroye, L. Györfi. A probabilistic theory of pattern recognition, springer. 1997.

[11] W. B. Langdon. The evolution of size in variable length representations. In *ICEC'98*, pages 633–638. IEEE Press, 1998.

[12] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In John Koza, editor, *Late Breaking Papers at GP'97*, pages 132–140. Stanford Bookstore, 1997.

[13] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[14] W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. Angeline, editors, *Advances in Genetic Programming III*, pages 163–190. MIT Press, 1999.

[15] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 April 1998.

[16] H.L. Lebesgue. *Intégrale, Longueur, Aire*. University of Nancy, 1902.

[17] Sean Luke and Liviu Panait. Lexicographic parsimony pressure. In W. B. Langdon et al., editor, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836. Morgan Kaufmann Publishers, 2002.

[18] P.L. Bartlett M. Antony. Neural network learning : Theoretical foundations, cambridge university press. 1999.

[19] Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 310–317, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[20] V. Pareto. *Manuale d'Economia Politica*. Milano: Societ Editrice, Libraria, 1906.

[21] S.D. Poisson. Recherche sur la probabilité des jugements, principalement en matière criminelle. *Comptes-Rendus hebdomadaires des Séances de l'Académie des Sciences*, 1:473–494, 1835.

[22] A. Ratle and M. Sebag. Avoiding the bloat with probabilistic grammar-guided genetic programming. In P. Collet et al., editor, *Artificial Evolution VI*. Springer Verlag, 2001.

[23] H. Rogers. *Theory of recursive functions and effective computability*. McGraw-Hill, New York, 1967.

[24] J. Schmidthuber. Hierarchies of generalized kolmogorov complexities and nonenumerable universal measures computable in the limit. *International Journal of Foundations of Computer Science 13(4):587-612*, 2002.

[25] Sara Silva and Jonas Almeida. Dynamic maximum tree depth : A simple technique for avoiding bloat in tree-based gp. In E. Cantú-Paz et al., editor, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1776–1787. Springer-Verlag, 2003.

[26] M. Sipser. A complexity theoretic approach to randomness. In *Proceedings of the 15th ACM Symposium on the Theory of Computing*, pages 330–335, 1983.

[27] Ray Solomonoff. A formal theory of inductive inference, part 1. *Inform. and Control, vol. 7, number 1, pp. 1-22*, 1964.

[28] Ray Solomonoff. A formal theory of inductive inference, part 2. *Inform. and Control, vol. 7, number 2, pp. 222-254*, 1964.

[29] T. Soule and J. A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309, 1998.

[30] Terence Soule. Exons and code growth in genetic programming. In James A. Foster et al., editor, *EuroGP 2002*, volume 2278 of *LNCS*, pages 142–151. Springer-Verlag, 2002.

[31] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *proceedings of the London Mathematical Society, Ser. 2, 45, pp. 161-228 (reprinted in M. Davis (1965), The Undecidable, Ewlett, NY: Raven Press, pp. 155-222)*, 1936-1937.

[32] V. Vapnik. The nature of statistical learning, springer. 1995.

[33] B.-T. Zhang and H. Muhlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation, vol. 3, no. 1, pp. 17-38*, 1995.